Algorithm Implementation

Dr. Edward Nava

ejnava@unm.edu

UNM | Electrical & Computer Engineering

Pseudocode

- **When documenting an algorithm in a language such as C, programmers use descriptive variable names such as: speed, volume, size, count, amount, etc. After the program is compiled, these variable names correspond to memory locations.**

- **To efficiently execute code, a compiler will attempt to keep the variables that are referenced most often in processor registers because access is much faster than access to memory.**

Pseudocode Guidelines

- When using pseudocode to document an assembly language program, you should use the names of the registers you intend to use in the assembly language code.
- It is advisable to create a cross reference table between the processor register name and what it is being used for in the program and comments should also document the usage.
- We use register names in pseudocode because the purpose of the pseudocode is to document the assembly language program.

Pseudocode Guidelines

- **You will find that as you develop the pseudocode, it is a simple process to translate pseudocode into assembly language code.**

- **For (t0=1; t0 < s0; t0++) do {this block of code};**

Pseudocode Details

- **Pseudocode for assembly language programs will have the appearance of C in terms of control structures and arithmetic expressions, but descriptive variable names will usually only appear in the LOAD ADDRESS (la) instruction where there is a reference to a symbolic memory address.**

- **In assembly language you define and allocate space for variables in the data segment of memory using assembler directives such as *.word* and *.space*.**

MIPS Assembly Language Syntax

[label:] Op-Code [operand], [operand], [operand] [#comment]

Label	Op-Code	Dest.	S1,	S2	Comment
chico:	**add**	**$a0,**	**$t0,**	**$t1**	**# a0 = t0 + t1**

Pseudocode to Assembly

Translation an Arithmetic Expression
$s0 = srt ($a0 * $a0 + $a1 * $a1)

Hypotenuse:

```
mult    $a0, $a0        # Square $a0
mflo    $t0             # t0 = Lower 32-bits of product
mult    $a1, $a1        # Square $a1
mflo    $t1             # t1 = Lower 32-bits of product
add     $a0, $t0, $t1   # a0 = t0 + t1
jal     srt             # Call the square root function
move    $s0, $v0        # By convention, the result of sqr
                        # is returned in $v0
```

Area of a Circle

$\$s0 = \pi * \$t8 * \$t8$

Area:

```
li $t0, 314159     # Load immediate Pi scaled up 100,000
mult $t8, $t8      # Radius squared
mflo $t1           # Move lower 32-bits of product in
                   # Low register to $t1
mult $t1, $t0      # Multiply by scaled Pi
mflo $s0           # Move lower 32-bits of product in
                   # Low register to $s0
li $t1, 100000     # Load immediate scale factor of 100,000
div $s0, $t1       # Divide by scale factor
mflo $s0           # Truncated integer result left in $s0
```

Translation of an "if … then … else …" Control Structure

if ($t8 < 0) then

{$s0 = 0 - $t8; $t1 = $t1 +1}

else

{$s0 = $t8; $t2 = $t2 + 1}

```
        bgez $t8, else          # if ($t8 is greater than or
                                # equal to zero) branch to else
        sub $s0, $zero, $t8     # $s0 gets the negative of $t8
        addi $t1, $t1, 1        # increment $t1 by 1
        b next                  # branch around the else code
else:
        move $s0, $t8           # $s0 gets a copy of $t8
        addi $t2, $t2, 1        # increment $t2 by 1
next:
```

UNM Electrical & Computer Engineering

Translation of a “while” Control Structure

while ($a1 < $a2) do
{ $a1 = $a1 + 1
$a2 = $a2 - 1}

```
while:
        bgeu $a1, $a2, done       # If( $a1 >= $a2) Branch to done
        addi $a1, $a1, 1          # $a1 = $a1 + 1
        addi $a2, $a2, -1         # $a2 = $a2 - 1
        b while                   # Branch to while
done:
```

Translation of a “for” Loop Control Structure

$a0 = 0;
for ($t0 =10; $t0 > 0; $t0 = $t0 -1) do
{$a0 = $a0 + $t0}

```
        li $a0, 0         # $a0 = 0
        li $t0, 10        # Initialize loop counter to 10
loop:
        add $a0, $a0, $t0
        addi $t0, $t0, -1 # Decrement loop counter
        bgtz $t0, loop    # If ($t0 > 0) Branch to loop
```

UNM | Electrical & Computer Engineering

Translation of a “switch” Control Structure

Typical structure using c++ types of I/O – (For illustration only.)

```
        $s0 = 32;
top:    cout << "Input a value from 1 to 3"
        cin >> $v0
        switch ($v0)
        { case(1):{$s0 = $s0 << 1; break;}
        case(2):{$s0 = $s0 << 2; break;}
        case(3):{$s0 = $s0 << 3; break;}
        default: goto top; }
        cout << $s0
```

```
            .data
            .align 2
jumptable:  .word top, case1, case2, case3
prompt:     .asciiz "\n\n Input a value from 1 to 3: "
            .text
top:
            li $v0, 4               # Code to print a string
            la $a0, prompt
            syscall
            li $v0, 5               # Code to read an integer
            syscall
            blez $v0, top           # Default for less than one
            li $t3, 3
            bgt $v0, $t3, top       # Default for greater than 3
            la $a1, jumptable       # Load address of jumptable
            sll $t0, $v0, 2         # Compute word offset (multiply by 4)
            add $t1, $a1, $t0       # Form a pointer into jumptable
            lw $t2, 0($t1)          # Load an address from jumptable
            jr $t2                  # Jump to specific case "switch"
case1:      sll $s0, $s0, 1         # Shift left logical one bit
            b output
case2:      sll $s0, $s0, 2         # Shift left logical two bits
            b output
case3:      sll $s0, $s0, 3         # Shift left logical three bits
```

```
output:
        li $v0, 1       # Code to print an integer is 1
        move $a0, $s0   # Pass argument to system in $a0
        syscall         # Output result
```

A Few Words on Overflow Detection

- **With 2's compliment numbers, when adding numbers of the same sign, the sum must have the same sign as the operands.**
- **A carry at the most significant bit does not signify that a signed overflow has occurred.**
- **With unsigned integers, when a carry occurs at the most significant bit, this is an unsigned overflow.**

Assembler Directives

- **Give the programmer the ability to establish some initial data structures that will accessed by the computer at run time.**
- **They begin with “.”**
- **Examples:**
  - .align – align the next datum on a 2^n byte boundary
  - .ascii string – store string in memory with no null termination
  - .asciiz string – store string with null termination
  - .data – items stored in the data segment
  - .text – items stored in the text segment

Assembler Directives

To allocate space in memory for a one-dimensional array of 1024 integers, the following construct is used in the C language:

```
int ARRAY[1024] ;
```

In MIPS assembly language, the corresponding construct is:

```
        .data
ARRAY: .space 4096
```

Assembler Directives

To initialize a memory array before program execution begins with a set of 16 values corresponding to the powers of 2 (2N with N going from 0 to 15) , the following construct is used in the C language:

Int Pof2[16] = { 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 }

In MIPS assembly language the corresponding construct is:

```
        .data
Pof2:   .word 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192,
        16384, 32768
```

Here is an example of how MIPS code can be written to access element two in the array and place a copy of the value in reg. $s0. The load address la macro instruction is used to initialize a pointer in "$a0" with the base address of the array labeled "Pof2."

```
la $a0, Pof2        # a0 = &Pof2
lw $s0, 8($a0)      # s0 = MEM[a0 + 8]
```